5/1/12
P.42

PREPRINT/ IN/31

The item to be cleared is a medium-fidelity software simulation model of a vented cryogenic tank. Such tanks are commonly used to transport cryogenic liquids such as liquid oxygen via truck, and have appeared on liquid-fueled rockets for decades.

This simulation model works with the HCC simulation system that was developed by Xerox PARC and NASA Ames Research Center. HCC has been previously cleared for distribution.

When used with the HCC software, the model generates simulated readings for the tank pressure and temperature as the simulated cryogenic liquid boils off and is vented. Failures (such as a broken vent valve) can be injected into the simulation to produce readings corresponding to the failure.

Release of this simulation will allow researchers to test their software diagnosis systems by attempting to diagnose the simulated failure from the simulated readings.

This model does not contain any encryption software nor can it perform any control tasks that might be export controlled.

```
//  This system consists of a liquid oxygen tank with a pneumatically
//  controlled vent valve.  As the temp rises, oxygen boils and the
//  pressure rises.  At a set pressure, the pneumatic valve should open
//  to release gaseous oxygen, decreasing the pressure.  When the pressure
//  is sufficiently low, the valve should close to prevent excessive
//  oxygen loss.
//
//  There are a number of failures that can be injected to alter the
//  simulated operation of the system.
//
//  The layout of the system is as follows:
//
//  The high pressure pneumatic tank is filled with an inert gas.  The pressure
//  from this tank is regulated down to a working pressure, and is used to
//  provide the force to open/close the pneumatic valve.  The temp and pressure
//  at the pneumatic tank are measured, as is the pressure of the regulated
//  pneumatic pressure.
//
//  The liquid oxygen tank is connected to the pneumatic valve.  When the valve is open,
//  gaseous oxygen that has boiled in the tank is vented to the atmosphere.  The oxygen
//  pressure in front of the valve and the temperature after the valve are sensed.
//  Since the gaseous oxygen is extremely cold, the temp sensor should show
//  a decreased temperature when oxygen is being vented.
//
//                                                                  Vent to Atmosphere
//                                                                          |
//                      |-temp sensor              pressure sensor-|       |
//                      |-pressure sensor                          |       |- temp sensor
//                      |                                          |       |
//  +-----------------+ |     +-----------+    +-----------+       |    +-----------------+
//  + high pressure  +==+=====+ Regulator +====+ Regulator +=======+====+ pneumatic valve+
//  + pneumatic tank +        +-----------+    +-----------+            +-----------------+
//  +-----------------+                                                          |
//                                                                 3 pressure sensors--|
//                                                                                    |
//                                                                    +---------------+
//                                                                    +               +
//                                                                    + oxygen tank   +
//                                                                    +               +
//                                                                    +---------------+
//
////////////////////////////////////////////////////////////////////////////////

#include "Flow.hcc"
#include "Tank.hcc"
#include "ActuatingSolenoidValve.hcc"
#include "PneumaticValve.hcc"
#include "Regulator.hcc"
#include "Orifice.hcc"

#include "LoxTank.hcc"
#include "Controller.hcc"
#include "Sensors.hcc"

%integration_init 0.001


////////////////////////////////////////////////////////////////////////////////
//
//  Objects for pneumatic system
```

```
always Flow TK02_SV08 = eval(new Flow());
always Flow SV08_RG21 = eval(new Flow());
always Flow RG21_RG02 = eval(new Flow());
always Flow RG02_SV31 = eval(new Flow());
always Flow SV31_Out = eval(new Flow());
always Flow SV31_OF01 = eval(new Flow());
always Flow OF01_VR01 = eval(new Flow());

always Tank TK02 = eval(new Tank(TK02_SV08, 29.24, 540, 8.47, 0.00882));

always SolenoidActuatingOnOffGasValve SV08 =
        eval(new SolenoidActuatingOnOffGasValve(TK02_SV08,SV08_RG21,1));

always Regulator RG21 = eval(new Regulator(SV08_RG21, RG21_RG02, 700));

always Regulator RG02 = eval(new Regulator(RG21_RG02, RG02_SV31, 750));

always SolenoidActuating3wayGasValve SV31 =
    eval(new SolenoidActuating3wayGasValve(RG02_SV31, SV31_Out,SV31_OF01, 1));

always Orifice OF01 = eval(new Orifice(SV31_OF01, OF01_VR01, 0.00545,
                        0.000341, 1.66, 1.0, 0.0088));

always {
        SV31_Out.p = 15;
        if (SV31_Out.flowsign > 0) {
            SV31_Out.t = 390;
        }
}


////////////////////////////////////////////////////////////////////////
//
// Objects for LOX tank and vent system
//
////////////////////////////////////////////////////////////////////////

always Flow InTank = eval(new Flow());
always Flow MainOut = eval(new Flow());
always Flow LO2_VR01 = eval(new Flow());
always Flow VR01_Out = eval(new Flow());

always PneumaticOnOffGasValve VR01 =
        eval(new PneumaticOnOffGasValve(LO2_VR01, VR01_Out, OF01_VR01,
             0.01, 0, 0.065, 2, 0.0705) );
//    public PneumaticOnOffGasValve(Flow In, Flow Out, Flow TankIn,
//      interval tank_mass_init, interval defaultState, interval diameter,
//      interval loss_coeff) {


always LoxTank LO2 = eval(new LoxTank(InTank, LO2_VR01, MainOut, 21400,
            162.25, 0.0705, 10, 30.56));

always Controller V1Control = eval(new Controller(SV31, LO2));

always {

    //LO2_VR01.p = 30;
    //LO2_VR01.t = 540;
```

```
    //MainOut.p = 3.5; // Pressure at 35,000 ft
    MainOut.mass_rate = 0.0;
}

////////////////////////////////////////////////////////////////////////
//
// Sensors (Pressure, temperature, microswitches)
//
////////////////////////////////////////////////////////////////////////

// Sensors on the pneumatic system.  Supply pressure, regulated pressure, temperature
always PressureSensor PneumSupP = eval(new PressureSensor(TK02_SV08,5,0,0.1));
always PressureSensor PneumRegP = eval(new PressureSensor(RG02_SV31,5,0,0.1));
always TemperatureSensor PneumT=eval(new TemperatureSensor(TK02_SV08,0,0.2));

// Sensors on the LOX vent.  Three redundant pressure sensors, plus temp
always PressureSensor VentPA = eval(new PressureSensor(LO2_VR01, 5, 0, 0.1));
always PressureSensor VentPB = eval(new PressureSensor(LO2_VR01, 5, 0, 0.1));
always PressureSensor VentPC = eval(new PressureSensor(LO2_VR01, 5, 0, 0.1));
always TemperatureSensor VentT=eval(new TemperatureSensor(VR01_Out,0,0.2));

// Sensor on the pneumatic cut-off.   This is usually never closed
always MicroswitchOnOffValve PneumCutPos = eval(new MicroswitchOnOffValve(SV08));

// Sensor on the Vent valve
always Microswitch3wayValve VentPos = eval(new Microswitch3wayValve(SV31));


////////////////////////////////////////////////////////////////////////
//
// Variables to sample
//
////////////////////////////////////////////////////////////////////////

//Needed for Labview:
//sample(LO2.p_gox, LO2.t_gox);

//Debugging sampling:
//sample (VR01.PVT.p, LO2_VR01.p, LO2_VR01.t, LO2_VR01.mass_rate);
//sample(LO2.p_gox, LO2.t_gox, LO2.v_gox, LO2.m_gox, LO2.m_boilrate, LO2.t_lox, LO2.m_lox,
LO2.p_sat);


////////////////////////////////////////////////////////////////////////
//
// Commands and faults
//
////////////////////////////////////////////////////////////////////////

//when (time = 100) SV08.SetClosed();
//when (time = 110) SV08.SetOpen();
when (time = 0.01) SV31.SetClosed();
//when (time = 170) SV31.SetOpen();
//when (time = 230) SV31.SetClosed();
//when (time = 270) SV31.SetOpen();

// Manual valve open/close commands, to see if
```

```
// With manual commands, runtime for 10000 = 54 realtime s
// With full controller,  runtime = over 20 min
// With controller without "Done" state = 56 s

//when (time = 1800) SV31.SetOpen();
//when (time = 2340) SV31.SetClosed();
//when (time = 3046) SV31.SetOpen();
//when (time = 3170) SV31.SetClosed();
//when (time = 3945) SV31.SetOpen();
//when (time = 4080) SV31.SetClosed();
//when (time = 4933) SV31.SetOpen();
//when (time = 5081) SV31.SetClosed();
//when (time = 6018) SV31.SetOpen();
//when (time = 6180) SV31.SetClosed();
//when (time = 7211) SV31.SetOpen();
//when (time = 7388) SV31.SetClosed();
//when (time = 8520) SV31.SetOpen();
//when (time = 8714) SV31.SetClosed();

// Fault injection commands
//when (time = 1000) RG21.SetNotRegulating();
//when (time = 1000) RG02.SetNotRegulating();

//when (time = 3000) SV31.SetStuckOpen(0);
//when (time = 1000) SV31.SetStuckClosed(0);
//when (time = 1000) VR01.SetStuckOpen(0);
//when (time = 3000) VR01.SetStuckClosed(0);

//when (time = 90) RG12.SetRegulatingHigh();
//when (time = 90) RG12.SetRegulatingLow();
//when (time = 90) RG12.SetNotRegulating();
//when (time = 90) RG12.SetBlocked();
//when (time = 100) RG12.ClearFault();
```

```
//////////////////////////////////////////////////////////////

#ifndef ActuatingValve_HEADER
#define ActuatingValve_HEADER

%module "valvefault"

#include "Common.hcc"
#include "Flow.hcc"

//////////////////////////////////////////////////////////
//  Solenoid Valve (Actuating, On-Off, Gas) object
//////////////////////////////////////////////////////////

class SolenoidActuatingOnOffGasValve {

    private interval p;
    private interval probClearFault;
    private boolean StuckOpen, StuckClosed, RemoveFault;

    public interval state, default_state, junk;

    public void SetOpen();
    public void SetClosed();
    public void SetStuckOpen(interval probclear);
    public void SetStuckClosed(interval probclear);
    public void ClearFault();

    public SolenoidActuatingOnOffGasValve(Flow In, Flow Out,
            interval defaultState) {

        //sample(state, density, mass_flow, StuckOpen);
        //sample(StuckClosed);

        storeval(state, default_state);

        always {
            default_state := eval(defaultState);

            // This type of valve only passes the flow information
            // through if it's open, and passes p = 0 if it's
            // closed.  There isn't any setting of the mass flowrate.

            Out.t := In.t;
            Out.mass_rate = In.mass_rate;

            unless (StuckClosed) {
              unless (StuckOpen) {
                unless ( (state = 0) || (In.p = 0) ) {
                    Out.p = In.p;
                } else { // ( (state = 0) || (In.p <= Out.p) || (In.p = 0) )//
                    Out.p = 0;
                }
              }
            }

            /* Define the two failure modes for the valve, StuckOpen
               and StuckClosed.  As you'd expect, StuckOpen will let
```

```
            and StuckClosed will not let the gas through. *
          if (StuckOpen) {
              unless ( In.p = 0 ) {
                  Out.p = In.p;
              }
              else /*( In.p = 0 ) */ {
                  Out.p = 0;
              }
          }

          if (StuckClosed) {
              Out.p := 0;
          }
      }
  }

  /* Define the access functions to turn the valves on
     and off.  Note that there will be a probability of unsticking
     the valve when we try to open and close it.  When the valve
     is stuck open, we can't close it, and sending the command
     to open again might free the valve.  Also, when the valve
     is stuck closed, we can't open it, and so sending the
     command to close again might free the valve.  */
  public void SetOpen() {
      storeval(state, 1);
      if (StuckOpen) {
          interval clear = check_to_clear_fault(probClearFault);
          if (clear = 1) ClearFault();
      }
  }

  public void SetClosed() {
      storeval(state, 0);
      if (StuckClosed) {
          interval clear = check_to_clear_fault(probClearFault);
          if (clear = 1) ClearFault();
      }
  }

  /* Define the access functions that allow the user to
     inject faults to the valve.  These are mutually exclusive,
     so SetStuckOpen will do nothing if we called a SetStuckClosed
     without clearing it (and vice versa). */
  public void SetStuckOpen(interval probclear)  {
      unless (StuckClosed) {
        unless (StuckOpen)   {
          do {
              hence StuckOpen;
          } watching (RemoveFault);
          storeval(probClearFault, probclear);
        }
      }
  }

  public void SetStuckClosed(interval probclear) {
      unless (StuckClosed) {
        unless (StuckOpen) {
          do {
              hence StuckClosed;
          } watching (RemoveFault);
          storeval(probClearFault, probclear);
```

```
    }

    public void ClearFault() {
        RemoveFault;
    }

    public void CreateValveFault() {
        interval faulttype, faultprob;
        faulttype = check_for_fault(default_state, state);
        if( faulttype = 1.0 ) {
            faultprob = get_prob_to_clear();
            SetStuckOpen(faultprob);
        }

        if( faulttype = 2.0 ) {
            faultprob = get_prob_to_clear();
            SetStuckClosed(faultprob);
        }
    }

}

////////////////////////////////////////////////////////
//  Solenoid Valve (Actuating, 3-way, Gas) object
////////////////////////////////////////////////////////

class SolenoidActuating3wayGasValve {

    private interval p;
    private interval probClearFault;
    private boolean StuckOpen, StuckClosed, RemoveFault;

    public interval state, default_state;

    public void SetOpen();
    public void SetClosed();
    public void SetStuckOpen(interval probclear);
    public void SetStuckClosed(interval probclear);
    public void ClearFault();

    public SolenoidActuating3wayGasValve(Flow A, Flow B, Flow C,
            interval defaultState) {

        //sample(state, density, mass_flow, StuckOpen);
        //sample(StuckClosed);

        storeval(state, default_state);

        always {
            default_state := eval(defaultState);

            // This type of valve only passes the flow information
            // through A->C if it's open, and passes B<-C if it's
            // closed.

            unless (StuckClosed) {
              unless (StuckOpen) {
                unless ( state = 0) {
                    C.p = A.p;
                    C.t = A.t;
```

```
                C.flowsign = A.flowsign;
                B.mass_rate = 0.0;

            } else { // ( state = 0 )//
                C.p = B.p;
                C.t = B.t;
                C.mass_rate = B.mass_rate;
                C.flowsign = B.flowsign;
                A.mass_rate = 0.0;
            }
            /*unless ( state = 1) {
                C.p = B.p;
                C.t = B.t;
                C.mass_rate = B.mass_rate;
                C.flowsign = B.flowsign;
                A.mass_rate = 0.0;
            } else { // ( state = 1 )//
                C.p = A.p;
                C.t = A.t;
                C.mass_rate = A.mass_rate;
                C.flowsign = A.flowsign;
                B.mass_rate = 0.0;
            }*/
          }
        }

        /* Define the two failure modes for the valve, StuckOpen
           and StuckClosed.  As you'd expect, StuckOpen will let
           gas through regardless of the commanded set value,
           and StuckClosed will not let the gas through. */
        if (StuckOpen) {
                C.p = A.p;
                C.t = A.t;
                C.mass_rate = A.mass_rate;
                C.flowsign = A.flowsign;
                B.mass_rate = 0.0;
        }


        if (StuckClosed) {
                C.p = B.p;
                C.t = B.t;
                C.mass_rate = B.mass_rate;
                C.flowsign = B.flowsign;
                A.mass_rate = 0.0;
        }
    }
}

/* Define the access functions to turn the valves on
   and off.  Note that there will be a probability of unsticking
   the valve when we try to open and close it.  When the valve
   is stuck open, we can't close it, and sending the command
   to open again might free the valve.  Also, when the valve
   is stuck closed, we can't open it, and so sending the
   command to close again might free the valve.  */
public void SetOpen() {
    storeval(state, 1);
    if (StuckOpen) {
        interval clear = check_to_clear_fault(probClearFault);
        if (clear = 1) ClearFault();
```

```
    public void SetClosed() {
        storeval(state, 0);
        if (StuckClosed) {
            interval clear = check_to_clear_fault(probClearFault);
            if (clear = 1) ClearFault();
        }
    }

    /* Define the access functions that allow the user to
       inject faults to the valve.  These are mutually exclusive,
       so SetStuckOpen will do nothing if we called a SetStuckClosed
       without clearing it (and vice versa). */
    public void SetStuckOpen(interval probclear)  {
        unless (StuckClosed) {
          unless (StuckOpen)   {
            do {
                hence StuckOpen;
            } watching (RemoveFault);
            storeval(probClearFault, probclear);
          }
        }
    }

    public void SetStuckClosed(interval probclear) {
        unless (StuckClosed) {
          unless (StuckOpen) {
            do {
                hence StuckClosed;
            } watching (RemoveFault);
            storeval(probClearFault, probclear);
          }
        }
    }

    public void ClearFault() {
        RemoveFault;
    }

    public void CreateValveFault() {
        interval faulttype, faultprob;
        faulttype = check_for_fault(default_state, state);
        if( faulttype = 1.0 ) {
            faultprob = get_prob_to_clear();
            SetStuckOpen(faultprob);
        }

        if( faulttype = 2.0 ) {
            faultprob = get_prob_to_clear();
            SetStuckClosed(faultprob);
        }
    }

}


/*********************************************************
  CreateValveFault()
 *********************************************************/
/* Used to enable the faults for the solenoid valves.  Will call
```

```
    to fill itself there is a constant (low) probability that
    the valve will become stuck.  This results in the valve faults
    having a geometric probability distribution.  May not be
    realistic, but the real point is to be able to add faults
    to the valves in order to test the control software!
  */
/*
//CreateValveFault(),  // Uncomment these lines to enable the random faults

void CreateValveFault() {
    SolenoidOnOffGasValve X;
    SolenoidSwitchingGasValve Y;

    // Check for faults in on-off solenoid valves
    forall X in class SolenoidOnOffGasValve {
        X.CreateValveFault();
    }

    // Check for faults in switch solenoid valves. Same code as
    //  above, but done for a different valve name.
    forall Y in class SolenoidSwitchingGasValve {
        Y.CreateValveFault();
    }

    when(time = eval(time)+1) CreateValveFault();
}
*/


#endif
```

```
#ifndef _common_HEADER
#define _common_HEADER

#include <storeval.hcc>

#define PI 3.14159

%EPSILON  1e-10

// Ideal gas constant.
//    8.31451 for Pa-m^3/mol-K
//    8.31451e-2 for bar-L/mol-K
//    2.365e-2 for  psi-ft^3 / mol-R
#define r 2.365e-2 // psi*ft^3 / mol*R


#endif
```

```
// Controller.hcc  Adam Sweet
//
//////////////////////////////////////
#ifndef Controller_HEADER
#define Controller_HEADER

#include "ActuatingSolenoidValve.hcc"
#include "LoxTank.hcc"

class Controller {
// There is one issue with the controller - when it contains the
// "StateFinished" wrapper, to end controlling the valve, after
// about six valve cycles, hcc begins to run Very slowly in the point
// phases.  We believe that the reason is because the do{}watching
// statement keeps the other when statements from being removed until
// "Done" becomes true.  I have therefore removed the "Done" state, and
// will try to implement something else.

// With manual commands, runtime for 10000s = 54 realtime s
// With full controller,  runtime = over 20 min
// With controller without "Done" state = 56 s

     private boolean Done;
     private SolenoidActuating3wayGasValve ValveIn;
     private LoxTank TankIn;

     private void StateClosed();
     private void StateVenting();
     private void StateFinished();

     public Controller(SolenoidActuating3wayGasValve V, LoxTank T) {

        always { ValveIn = eval'V); TankIn = eval(T); }

        do {
            when (time = 1800) StateVenting();
        } watching (Done);

        when (time = 10000) StateFinished();
     }

     private void StateVenting() {
        ValveIn.SetOpen();
        when (TankIn.p_gox <= 12.2) StateClosed();
     }

     private void StateClosed() {
        ValveIn.SetClosed();
        when (TankIn.p_gox >= 18) StateVenting();
     }

     private void StateFinish :d() {
        Done;
        ValveIn.SetClosed();
     }
}

#endif
```

```
#ifndef Flow_HEADER
#define Flow_HEADER

/*****************************************************************************
    Flow object
*****************************************************************************/

/* Used to connect the different components of the system together.
   Should not think of this so much as the pipes of the system
   as a software gimmick used to transfer flow between the components.
*/

#include "Common.hcc"

class Flow {
    public interval t, p, mass_rate, mw;
    public interval flowsign;

    /*public Flow(interval mw_init){
        storeval(mw, mw_init);
    }*/
}

#endif
```

```
*
*  LoxTank.hcc      Adam Sweet
*
*************************************************************/
/* File that implements the LOX tank on board the X-34.*/

#ifndef LoxTank_HEADER
#define LoxTank_HEADER


%module "thermo"

#include "Common.hcc"
#include "Flow.hcc"

class LoxTank {

   public interval v_tank, v_gox, v_lox;
   public interval p_gox, p_lox;
   public interval m_gox, m_lox, m_boilrate;
   public interval t_gox, t_lox;
   private interval u_gox, h_gox_in, h_gox_out;
   private interval mw_gox;

   private interval q_gox, q_lox, q_total;

   public interval vapor_pressure, p_sat;


   public LoxTank(Flow Input,
            Flow VentOutput, Flow MainOutput, interval m_lox0,
            interval t_lox0, interval mw_gox0,
            interval area, interval length) {
         // m_lox0 = lbm, t_lox0 = deg R, mw_gox0 = lbm/mol,
         // area = ft^2, length = ft

         //sample(m_gox, t_gox, v_gox, p_gox, p_sat, t_lox, v_lox, m_boilrate);

         //sample(p_sat, m_boilrate);

         /* Set tank initial status */
         m_lox := eval(m_lox0);
         t_lox := eval(t_lox0);
         u_gox := get_o2_u_from_T(t_lox);
         //u_gox := get_o2_u_from_T(450);
         always {mw_gox := eval(mw_gox0);
                 v_tank := eval(area*length);}

         /* Determine the vapor pressure of the LOX, to determine
            the initial mass of GOX in the tank. */
         //vapor_pressure =-461.14+10.286*t_lox-0.0778*t_lox^2+0.0002*t_lox^3;
         m_gox := (1.1*p_sat*v_gox*mw_gox)/(r*t_gox);
         //m_gox := 1.54;

         always {

             /* Determine the volume of LOX in the tank.  Use that to
                find the volume in the tank for gas.  */
             v_lox := (1/71.5)*m_lox;  //density LOX = 71.5 lbm/ft^3
             v_gox := v_tank - v_lox;
```

```
        // 3rd order fit:
            //p_sat := 512.738 - 7.05063*t_lox + 0.024537*t_lox^2;
        // 5th order fit:
            p_sat := 272.968 - 9.83445*t_lox + 0.139169*t_lox^2
              - 9.4813e-4*t_lox^3 + 2.9745e-6*t_lox^4 - 3.00628e-9*t_lox^5;

        /*unless ( p_gox < p_sat)
            m_boilrate := 0;
        else
            m_boilrate := 0;*/

        unless ( p_gox <= p_sat)
            m_boilrate := 0;
        else
            m_boilrate := VentOutput.mass_rate
                    + VentOutput.mass_rate*0.5*(p_sat-p_gox) - 0.001;
            //m_boilrate := 0.5*(p_sat-p_gox) - 0.001;

        /***** Conservation of mass for tank *****/
        m_lox' := -m_boilrate;
        m_gox' := m_boilrate - VentOutput.mass_rate;

        /**** Conservation of energy for tank *****/
            //91.5 = heat of vaporization of lox
            //0.4 = Cp of lox
        q_total := 9.8; //BTU/s
        //q_lox := 0.995*q_total;   //v_gox/(v_gox+v_lox)*q_total;
        //q_gox := 5.0e-3*q_total; //v_lox/(v_gox+v_lox)*q_total;
        q_gox := 1.75e-5*(550-t_gox)*q_total;
        q_lox := q_total - q_gox;

        t_lox' := (-91.5*m_boilrate + q_lox)/(0.4*m_lox);

        h_gox_in := get_o2_h_from_T(t_lox);
        h_gox_out := get_o2_h_from_T(t_gox);

        u_gox' := 1/m_gox * (m_boilrate*h_gox_in
                    - VentOutput.mass_rate*h_gox_out + q_gox
                    - m_gox'*u_gox);
        t_gox := get_o2_T_from_u(u_gox);
        //t_gox = t_lox;


        /***** Equation of state for gas in tank ******/
        p_gox := (m_gox/mw_gox) * r * t_gox / v_gox;
        //p_gox' := (m_gox'/mw_gox) * r * t_gox / v_gox
                    //+ (m_gox/mw_gox) * r * t_gox' / v_gox;

        /****** Set outgoing variables **************/
        VentOutput.p = p_gox;
        VentOutput.t = t_gox;
        VentOutput.mw = mw_gox;


    }

  } // end constructor

} // end class definition

#endif
```

```
// Orifice.hcc    Adam Sweet
//////////////////////////////////////////////////////
#ifndef Orifice_HEADER
#define Orifice_HEADER

%module "mathlib"

#include "Common.hcc"
#include "Flow.hcc"

// Orifice equations are taken from Marks' Handbook for Mechanical
// Engineers, p. 4-21.  They use the units:
// Velocity = ft/s
// Area = ft^2
// pressure = lbf / ft^2
// temperature = Rankine
// mass flowrate = lbm/s
// Ideal gas constant = (lbf/ft^2*ft^3)/(lbm/Rankine)
//    Note that they use the R specific to one gas.
//    I will be converting my R to that value by
//    dividing R by the molecular weight of the gas.  (lbm/mol)
// There is also another constant listed in the equation, gc.  This
// is a dimensional conversion constant from Bernoulli's equation,
// effectively equal to
// gc = (lbm-ft/s^2 *ft)/2*Btu = 1/64.4*(lbf-ft/Btu) = 12.1
// I will be adding my own constants into the orifice equation, to
// convert pressure from psi to lbf/ft^2, etc.



class Orifice {
    private interval mass_rate;
    private interval A1, A2, k;  // A1 = large dia, A2 = small dia,
    private interval C, mw;       // k = ratio of specific heats
                                  // C = discharge coeffecient
                                  // mw = molecular weight

    private interval term1, term2, num, denom;
    private interval flowsign;

    public Orifice(Flow In, Flow Out, interval area1, interval area2,
                   interval k0, interval C0, interval mw0) {

        storeval(A1, eval(area1));
        storeval(A2, eval(area2));
        storeval(k, eval(k0));
        storeval(C, eval(C0));
        storeval(mw, eval(mw0));

        //sample(In.p, In.t, mass_rate, Out.p, Out.t);

        always {

            In.mass_rate = mass_rate;
            Out.mass_rate = mass_rate;
            In.t = Out.t;
            In.flowsign = flowsign;
            Out.flowsign = flowsign;
```

```
unless ((In.p < Out.p) || (In.p = 0)) { // Flow in normal dir
    unless (0.5*In.p > Out.p) {
       // Normal flow
       term1:= (In.p/Out.p)^((k-1)/k)*((In.p/Out.p)^((k-1)/k) - 1);
       term2 := (2*12.1*k)/(r/mw*In.t*(k-1));
       num := C*A2*Out.p*sqrtPt(term1*term2);
       denom := sqrtPt(1 - (A2/A1)^2*(Out.p/In.p)^(2/k) );
       mass_rate := num/denom;

       //mass_rate = 0.0000001*(In.p - Out.p)*In.t;
    } else {
       // Choked flow
       term1 = k * (2/(k+1))^((k+1)*(k-1));
       term2 = 12.1/(144*r/mw*In.t);
       mass_rate = C*A2*(144*In.p)*sqrtPt(term1*term2);
       //mass_rate = 0.0295*In.p*sqrtPt(0.0315/In.t);
       //mass_rate = 0.001*(In.p - Out.p)*sqrtPt(1/In.t); // Works
       //mass_rate := 0.01*(In.p);  // Doesn't work
    }
    flowsign = 1.0;
}
if ( (In.p < Out.p) && (In.p != 0)) { // Flow in opposite dir
    unless (0.5*Out.p > In.p) {
       // Normal flow
       term1:= (Out.p/In.p)^((k-1)/k)*((Out.p/In.p)^((k-1)/k) - 1);
       term2 := (2*12.1*k)/(r/mw*Out.t*(k-1));
       num := C*A2*In.p*sqrtPt(term1*term2);
       denom := sqrtPt(1 - (A2/A1)^2*(In.p/Out.p)^(2/k) );
       mass_rate := - num/denom;
       //mass_rate = -0.0000001*(Out.p - In.p)*Out.t;
    } else {
       // Choked flow
       term1 = k * (2/(k+1))^((k+1)*(k-1));
       term2 = 12.1/(144*r/mw*Out.t);
       mass_rate = -C*A2*(144*Out.p)*sqrtPt(term1*term2);
       //mass_rate = -0.0295*Out.p*sqrtPt(0.0315/Out.t);
       //mass_rate = -0.001*(Out.p - In.p)*sqrtPt(1/Out.t); //Works
       //mass_rate := -0.01*(Out.p); // Doesn't work
    }
    flowsign = -1.0;

}
if (In.p = 0) {
    mass_rate = 0.0;
    flowsign = 1.0;

}

      }
   }
}



#endif
```

```
/*  PneumaticValve.hcc    Adam Sweet

//////////////////////////////////////////////////////////
#ifndef PneumaticValve_HEADER
#define PneumaticValve_HEADER


#include "Common.hcc"
#include "Flow.hcc"

%module "mathlib"
%module "thermo"


class PneumaticValveTank {
    public interval mass, v, p, t, mw_gas;
    public interval input_enthalpy;

    public interval heat_flow;
    private interval flowsign_in;

    public PneumaticValveTank(Flow In, interval m0,
                interval t0, interval v0, interval mw_gas0) {
        mass := eval(m0);
        t := eval(t0);
        always { v:=eval(v0); mw_gas = eval(mw_gas0); }

        always {

            input_enthalpy = get_he_h_from_T(t);
            heat_flow = 0.0;
            flowsign_in = In.flowsign;

            /* Mass balance */
            mass' := In.mass_rate;

            /* Equation of state, used to determine overall pressure. */
            p := (mass / mw_gas)*r*t / v;

            /* Energy balance */
            t' := (1/(13.2 + 3.125*mass) )*(In.mass_rate*input_enthalpy
                - heat_flow);

                /* Energy balance based on internal energy */
                /*u' = 1/m * (Input.mass_rate*input_enthalpy
                   - HeatOutput.heat_flow - m'*u),
                t = get_he_T_from_u(u)*/


            /* Send pressure out to adjoining flows*/
            In.p := p;

            unless (flowsign_in > 0) {
                In.t = t;
            }


        }
    }
}
```

```
Pneumatic On/Off Gas Valve
//////////////////////////////////////////////////////////
class PneumaticOnOffGasValve {
    private interval dia, kl, density, mw, p_in, t_in;
    private interval volume_flow, mass_flow;
    private interval p, sqrt_term;
    private interval probClearFault;
    private boolean StuckOpen, StuckClosed, RemoveFault;

    private interval state, default_state;

    public PneumaticValveTank PVT;

    public void SetOpen();
    public void SetClosed();
    public void SetStuckOpen(interval probclear);
    public void SetStuckClosed(interval probclear);
    public void ClearFault();

    public PneumaticOnOffGasValve(Flow In, Flow Out, Flow TankIn,
        interval tank_mass_init, interval defaultState, interval diameter,
        interval loss_coeff, interval mw0) {

        //sample(state, p_in, t_in, density, mass_flow);

        //storeval(state, default_state);
        storeval(mw, mw0);

        always PVT = eval(new PneumaticValveTank(TankIn, tank_mass_init,
                540, 0.04, 0.00882));

        always {
            default_state := eval(defaultState);
            dia := eval(diameter);
            kl := eval(loss_coeff);


            unless (PVT.p >= 410) {
                state = 0;
            } else {
                state = 1;
            }

            /* We are assuming that the flow only goes in one
               direction through the valve. If the output pressure is
               greater than the input pressure, set the density to 0,
               to indicate no reverse flow through the valve. */

            /* Define the normal mode of operation for the valve.  It can
               open and close, and the mass flowrate through the valve
               depends on the geometry (Kl) and the pressure difference on both
               sides.
               Note that state indicates the state of the valve.
               Valve open:   state = 1
               Valve closed: state = 0
               This flag is used throughout the code.   */

            /* The mass flowrate is sent up to the tank ahead of the
               valve.  It is always a positive number.    */
            In.mass_rate := mass_flow;
```

```
        p_in = In.p;
        t_in = In.t;

        density := (In.p * mw) / (r*In.t);  // Units: lbm/ft^3 //
        p := In.p - Out.p;
        //Out.t := In.t;

        unless (StuckClosed) {
          unless (StuckOpen) {
            unless ( (state = 0) || (In.p <= Out.p) || (In.p = 0) ) {
                sqrt_term := sqrtPt( 2*(In.p - Out.p)*4636.8/(density*kl));
                volume_flow := (1/4) * PI * dia^2 * sqrt_term;
                mass_flow := volume_flow * density;
                Out.t := In.t;
            } else { // ( (state = 0) || (In.p <= Out.p) || (In.p = 0) )//
                sqrt_term = 0.0;
                volume_flow := 0.0;
                mass_flow := 0.0;
                Out.t = 400;
            }
          }
        }

        /* Define the two failure modes for the valve, StuckOpen
           and StuckClosed.  As you'd expect, StuckOpen will let
           gas through regardless of the commanded set value,
           and StuckClosed will not let the gas through. */
        if (StuckOpen) {
            unless ( (In.p <= Out.p) || (In.p = 0) ) {
                sqrt_term := sqrtPt( 2*(In.p - Out.p)*4636.8/(density*kl));
                volume_flow := (1/4) * PI * dia^2 * sqrt_term;
                mass_flow := volume_flow * density;
                Out.t := In.t;
            }
            else /*( (In.p <= Out.p) || (In.p = 0) ) */ {
                mass_flow := 0;
                Out.t = 400;
            }
        }


        if (StuckClosed) {
            mass_flow := 0;
            Out.t = 400;
        }
    }
}

/* Define the access functions to turn the valves on
   and off.  Note that there will be a probability of unsticking
   the valve when we try to open and close it.  When the valve
   is stuck open, we can't close it, and sending the command
   to open again might free the valve.  Also, when the valve
   is stuck closed, we can't open it, and so sending the
   command to close again might free the valve.  */
public void SetOpen() {
    storeval(state, 1);
    if (StuckOpen) {
        interval clear = check_to_clear_fault(probClearFault);
        if (clear = 1) ClearFault();
```

```
    }

    public void SetClosed() {
        storeval(state, 0);
        if (StuckClosed) {
            interval clear = check_to_clear_fault(probClearFault);
            if (clear = 1) ClearFault();
        }
    }

    /* Define the access functions that allow the user to
       inject faults to the valve.  These are mutually exclusive,
       so SetStuckOpen will do nothing if we called a SetStuckClosed
       without clearing it (and vice versa). */
    public void SetStuckOpen(interval probclear)  {
        unless (StuckClosed) {
          unless (StuckOpen)   {
            do {
                hence StuckOpen;
            } watching (RemoveFault);
            storeval(probClearFault, probclear);
          }
        }
    }

    public void SetStuckClosed(interval probclear) {
        unless (StuckClosed) {
          unless (StuckOpen) {
            do {
                hence StuckClosed;
            } watching (RemoveFault);
            storeval(probClearFault, probclear);
          }
        }
    }

    public void ClearFault() {
        RemoveFault;
    }

    public void CreateValveFault() {
        interval faulttype, faultprob;
        faulttype = check_for_fault(default_state, state);
        if( faulttype = 1.0 ) {
            faultprob = get_prob_to_clear();
            SetStuckOpen(faultprob);
        }

        if( faulttype = 2.0 ) {
            faultprob = get_prob_to_clear();
            SetStuckClosed(faultprob);
        }
    }



endif
```

```
/*////////////////////////////////////////////////////////////////////////*/

#ifndef Regulator_HEADER
#define Regulator_HEADER

#include <storeval.hcc>

#include "Flow.hcc"

class Regulator {

    private interval setpoint, p_in;
    private boolean RegHigh, RegLow, Blocked, UnRegulated, RemoveFault;

    public void SetRegulatingHigh();
    public void SetRegulatingLow();
    public void SetNotRegulating();
    public void SetBlocked();
    public void ClearFault();
    private interval Min(interval a, interval b);

    public Regulator(Flow Input, Flow Output, interval setpoint_init) {

        storeval(setpoint, eval(setpoint_init));

        //sample (UnRegulated, p_in);

        always {

          p_in = Input.p;

          unless (RegHigh || RegLow || UnRegulated || Blocked ) {
             Output.p := Min(Input.p, setpoint);
          }
          if (RegHigh) {
             Output.p := Min(Input.p, setpoint*1.5);
          }
          if (RegLow) {
             Output.p := Min(Input.p, setpoint/1.5);
          }
          if (UnRegulated) {
             Output.p := Input.p;
          }
          if (Blocked) {
             Output.p := 0.0;
          }

          Output.t = Input.t;
          Output.mass_rate = Input.mass_rate;

        }
    }

    public void SetRegulatingHigh() {
        unless (RegLow || UnRegulated || Blocked) {
           do {
              always RegHigh;
           } watching (RemoveFault);
```

```
        }
    }

    public void SetRegulatingLow() {
        unless (RegHigh || UnRegulated || Blocked) {
            do {
                always RegLow;
            } watching (RemoveFault);

        }
    }

    public void SetNotRegulating() {
        unless (RegHigh || RegLow || Blocked) {
            do {
                always UnRegulated;
            } watching (RemoveFault);

        }
    }

    public void SetBlocked() {
        unless (RegHigh || RegLow || UnRegulated) {
            do {
                always Blocked;
            } watching (RemoveFault);

        }
    }

    public void ClearFault() {
        RemoveFault;
    }

    private interval Min(interval a, interval b) {

        unless ( a > b ) {
            return = a;
        } else {
            return = b;
        }

    }

}

#endif
```

```
// Sensors.hcc              Adam Sweet  6/12/00
//////////////////////////////////////////////////////////////////
#ifndef Sensors_HEADER
#define Sensors_HEADER

%module "mathlib"

#include <storeval.hcc>
#include "Flow.hcc"
#include "ActuatingSolenoidValve.hcc"
#include "LoxTank.hcc"

//////////////////////////////////////////////////////
//  Pressure sensor.
//////////////////////////////////////////////////////
// Reports the pressure of the input flow //
class PressureSensor {

    private interval p;
    private interval pmax;
    private interval mean, stdev;

    private boolean Unpowered;
    private boolean Shorted;
    private boolean RemoveFault;

    public void SetUnpowered();
    public void SetShorted();
    public void ClearFault();

    public PressureSensor(Flow Input, interval maxoutput,
                          interval noise_mean, interval noise_standard_dev) {
        storeval(pmax, maxoutput);
        storeval(mean, noise_mean);
        storeval(stdev, noise_standard_dev);

        sample(p);

        always {

           unless (Unpowered) {
             unless (Shorted)  {
                p := Input.p + gaussian(mean, stdev);
             }
           }

           if (Unpowered)
                { p := 0 + gaussian(mean, stdev); }
           if (Shorted)
                {p := pmax + gaussian(mean, stdev); }

        }

    } // end PressureSensor constructor

    public void SetUnpowered() {
        unless (Unpowered) {
          unless (Shorted)  {
                do {
```

```
                } until (RemoveFault);
            }
        }
    } // end SetUnpowered

    public void SetShorted() {
        unless (Unpowered) {
          unless (Shorted)  {
                do {
                    hence Shorted;
                } until (RemoveFault);
          }
        }
    } // end SetShorted

    public void ClearFault() {
        RemoveFault;
    } // end ClearFault

}



////////////////////////////////////////////////////
//  Temperature sensor.
////////////////////////////////////////////////////
// Reports the temperature of the input Flow //

class TemperatureSensor {

    private interval t;
    private interval mean, stdev;

    private boolean Broken;
    private boolean RemoveFault;

    public void SetBroken();
    public void ClearFault();

    public TemperatureSensor(Flow Input, interval noise_mean,
                             interval noise_standard_dev) {

        storeval(mean, eval(noise_mean));
        storeval(stdev, eval(noise_standard_dev));

        sample (t);

        always {
           unless (Broken)
                t := Input.t + gaussian(mean, stdev);
           if (Broken)
                t := 0.0 + gaussian(mean, stdev);
        }
    }

    public void SetBroken() {
        unless (Broken) {
            do {
                hence Broken;
            } until (RemoveFault);
        }
```

```
    public void ClearFault() {
        RemoveFault;
    } // end ClearFault

} // end Temperature Sensor class

////////////////////////////////////////////////
// Microswitches
////////////////////////////////////////////////
// Reports the open/closed status of a valve //
class MicroswitchOnOffValve {

    private interval state;

    private boolean StuckOn;
    private boolean StuckOff;
    private boolean RemoveFault;

    public void SetStuckOn();
    public void SetStuckOff();
    public void ClearFault();

    public MicroswitchOnOffValve(SolenoidActuatingOnOffGasValve V) {

        sample(state);

        always {
          unless (StuckOn) {
            unless (StuckOff) {
              state := V.state;
            }
          }
          if (StuckOn) {
              state := 1;
          }
          if (StuckOff) {
              state := 0;
          }
        }
    }

    public void SetStuckOn() {
        unless (StuckOff) {
            do {
                hence StuckOn;
            } until (RemoveFault);
        }
    } // end SetStuckOn

    public void SetStuckOff() {
        unless (StuckOn) {
            do {
                hence StuckOff;
            } until (RemoveFault);
        }
    } // end SetStuckOff

    public void ClearFault() {
            RemoveFault;
    } // end ClearFault
```

```
}

class Microswitch3wayValve {

    private interval state;

    private boolean StuckOn;
    private boolean StuckOff;
    private boolean RemoveFault;

    public void SetStuckOn();
    public void SetStuckOff();
    public void ClearFault();

    public Microswitch3wayValve(Solenoid Actuating3wayGasValve V) {

        sample(state);

        always {
          unless (StuckOn) {
            unless (StuckOff) {
              state := V.state;
            }
          }
          if (StuckOn) {
              state := 1;
          }
          if (StuckOff) {
              state := 0;
          }
        }
    }

    public void SetStuckOn() {
        unless (StuckOff) {
            do {
                hence StuckOn;
            } until (RemoveFault);
          }
    } // end SetStuckOn

    public void SetStuckOff() {
        unless (StuckOn) {
            do {
                hence StuckOff;
            } until (RemoveFault);
          }
    } // end SetStuckOff

    public void ClearFault() {
            RemoveFault;
    } // end ClearFault



////////////////////////////////////////////////////
/  LOXTemperature sensor.
////////////////////////////////////////////////////
/ Reports the temperature of the liquid 02 in the input LOX tank //
```

```
    private interval t;
    private interval mean, stdev;

    private boolean Broken;
    private boolean RemoveFault;

    public void SetBroken();
    public void ClearFault();

    public LOXTemperatureSensor(LoxTank Input, interval noise_mean,
                                interval noise_standard_dev) {
            // Arguments: Integer for sample order, input flow //

        storeval(mean, eval(noise_mean));
        storeval(stdev, eval(noise_standard_dev));
        sample (t);

        always {
           unless (Broken)
                t := Input.t_lox + gaussian(mean, stdev);
           if (Broken)
                t := 0.0 + gaussian(mean, stdev);
        }
    }

    public void SetBroken() {
        unless (Broken) {
            do {
                hence Broken;
            } until (RemoveFault);
        }
    } // end SetBroken

    public void ClearFault() {
        RemoveFault;
    } // end ClearFault

}  // end LOXTemperature Sensor class

#endif
```

```
#ifndef Tank_HEADER
#define Tank_HEADER


#include "Common.hcc"
#include "Flow.hcc"

%module "thermo"

class Tank {
    public interval mass, v, p, t, mw_gas;
    public interval input_enthalpy;

    public interval heat_flow;

    public Tank(Flow Out, interval m0,
                interval t0, interval v0, interval mw_gas0) {
        mass := eval(m0);
        t := eval(t0);
        always { v:=eval(v0); mw_gas = eval(mw_gas0); }

        always {

            heat_flow = 0;
            input_enthalpy = get_he_h_from_T(t);

            // Mass balance
            mass' := -Out.mass_rate;

            // Equation of state, used to determine overall pressure.
            p := (mass / mw_gas)*r*t / v;

            // Energy balance
            t' := (1/(13.2 + 3.125*mass) )*(-Out.mass_rate*input_enthalpy
                - heat_flow);

                // Energy balance based on internal energy
                /*u' = 1/m * (Input.mass_rate*input_enthalpy
                   - HeatOutput.heat_flow - m'*u),
                t = get_he_T_from_u(u)*/


            // Send pressure out to adjoining flows
            Out.p := p;
            Out.t := t;


        }
    }
}



#endif
```

```
#ifndef Valve_HEADER
#define Valve_HEADER

%module "mathlib"
%module "valveFault"

#include "Common.hcc"
#include "Flow.hcc"

/*********************************************************
  Solenoid Valve (On-Off, Gas) object
*********************************************************/

class SolenoidOnOffGasValve {
    private interval dia, kl, density, p_in, t_in;
    private interval volume_flow, mass_flow;
    private interval p, sqrt_term;
    private interval probClearFault;
    private boolean StuckOpen, StuckClosed, RemoveFault;

    private interval state, default_state;

    public void SetOpen();
    public void SetClosed();
    public void SetStuckOpen(interval probclear);
    public void SetStuckClosed(interval probclear);
    public void ClearFault();

    public SolenoidOnOffGasValve(Flow In, Flow Out,
            interval defaultState, interval diameter, interval loss_coeff) {

        //sample(state, density, mass_flow, StuckOpen);

        storeval(state, default_state);

        always {
            default_state := eval(defaultState);
            dia := eval(diameter);
            kl := eval(loss_coeff);


            /* We are assuming that the flow only goes in one
               direction through the valve. If the output pressure is
               greater than the input pressure, set the density to 0,
               to indicate no reverse flow through the valve. */

            /* Define the normal mode of operation for the valve.  It can
               open and close, and the mass flowrate through the valve
               depends on the geometry (Kl) and the pressure difference on both
               sides.
               Note that state indicates the state of the valve.
               Valve open:   state = 1
               Valve closed: state = 0
               This flag is used throughout the code.   */

            /* The molar flowrate is sent up to the tank ahead of the
               valve.  It is always a positive number.  The upstream tank
               then determines the mole fraction of the flow, and sends
               that result to the valve to be passed on down the system.  */
            In.mass_rate = mass_flow;
            Out.mass_rate = mass_flow;
```

```
        p_in = In.p;
        t_in = In.t;

        density := (In.p * In.mw) / (r*In.t) * 1000;  /* Set to kg/m^3*
        p := In.p - Out.p;
        Out.t := In.t;

        unless (StuckClosed) {
          unless (StuckOpen) {
            unless ( (state = 0) || (In.p <= Out.p) || (In.p = 0) ) {
                sqrt_term := sqrtPt( 2*(In.p - Out.p)*1.0e5/(density*kl) );
                volume_flow := (1/4) * PI * dia^2 * sqrt_term;
                mass_flow := volume_flow * density;
            } else { // ( (state = 0) || (In.p <= Out.p) || (In.p = 0) )//
                sqrt_term = 0.0;
                volume_flow := 0.0;
                mass_flow := 0.0;
            }
          }
        }

        /* Define the two failure modes for the valve, StuckOpen
           and StuckClosed.  As you'd expect, StuckOpen will let
           gas through regardless of the commanded set value,
           and StuckClosed will not let the gas through. */
        if (StuckOpen) {
            unless ( (In.p <= Out.p) || (In.p = 0) ) {
                sqrt_term := sqrtPt( 2*(In.p - Out.p)*1.0e5/(density*kl) );
                volume_flow := (1/4) * PI * dia^2 * sqrt_term;
                mass_flow := volume_flow * density;
            }
            else /*( (In.p <= Out.p) || (In.p = 0) ) */ {
                mass_flow := 0;
            }
        }


        if (StuckClosed) {
            mass_flow := 0;
        }
    }
}

/* Define the access functions to turn the valves on
   and off.  Note that there will be a probability of unsticking
   the valve when we try to open and close it.  When the valve
   is stuck open, we can't close it, and sending the command
   to open again might free the valve.  Also, when the valve
   is stuck closed, we can't open it, and so sending the
   command to close again might free the valve.  */
public void SetOpen() {
    storeval(state, 1);
    if (StuckOpen) {
        interval clear = check_to_clear_fault(probClearFault);
        if (clear = 1) ClearFault();
    }
}

public void SetClosed() {
    storeval(state, 0);
    if (StuckClosed) {
```

```
        if (clear = 1) ClearFault();
      }
    }

    /* Define the access functions that allow the user to
       inject faults to the valve.  These are mutually exclusive,
       so SetStuckOpen will do nothing if we called a SetStuckClosed
       without clearing it (and vice versa). */
    public void SetStuckOpen(interval probclear)  {
        unless (StuckClosed) {
          unless (StuckOpen)   {
            do {
                hence StuckOpen;
            } watching (RemoveFault);
            storeval(probClearFault, probclear);
          }
        }
    }

    public void SetStuckClosed(interval probclear) {
        unless (StuckClosed) {
          unless (StuckOpen) {
            do {
                hence StuckClosed;
            } watching (RemoveFault);
            storeval(probClearFault, probclear);
          }
        }
    }

    public void ClearFault() {
        RemoveFault;
    }

    public void CreateValveFault() {
        interval faulttype, faultprob;
        faulttype = check_for_fault(default_state, state);
        if( faulttype = 1.0 ) {
            faultprob = get_prob_to_clear();
            SetStuckOpen(faultprob);
        }

        if( faulttype = 2.0 ) {
            faultprob = get_prob_to_clear();
            SetStuckClosed(faultprob);
        }
    }

}

/*******************************************************
  Solenoid Valve (Switching, Gas) object
*************** ******************************************/

class SolenoidSwitchingGasValve {
    private interval state, default_state;
    private boolean StuckA, StuckB, RemoveFault;
    private interval probClearFault;

    public void SetA();
    public void SetB();
```

```
public void setStuckB(interval probToSet);
public void clearFault();

public SolenoidSwitchingGasValve(Flow In, Flow OutA, Flow OutB,
        interval defaultState) {
    always default_state = eval(defaultState);
    storeval(state, default_state);

    always {

        /* We are assuming that the flow only goes in one
           direction through the valve. If the output pressure is
           greater than the input pressure, set the density to 0,
           to indicate no reverse flow through the valve. */

        /* Define the normal mode of operation for the valve.  It can
           open and close, and the mass flowrate through the valve
           depends on the geometry (Kl) and the pressure difference on both
           sides.
           Note that state indicates the state of the valve.
               Valve output to A:    state = 0
               Valve output to B:    state = 1
           This flag is used throughout the code.    */

        /* The molar flowrate is sent up to the tank ahead of the
           valve.  It is always a positive number.  The upstream tank
           then determines the mole fraction of the flow, and sends
           that result to the valve to be passed on down the system.   */

        unless (StuckA){
          unless (StuckB) {
            unless (state = 1) {
                OutA.p = In.p;
                OutA.t = In.t;
                In.mass_rate = OutA.mass_rate;
                OutB.p = 0.0;
                OutB.t = In.t;
            } else /* state=1 */ {
                OutB.p = In.p;
                OutB.t = In.t;
                In.mass_rate = OutB.mass_rate;
                OutA.p = 0.0;
                OutA.t = In.t;
              }
          }
        }

        /* Define the two failure modes for the valve, StuckA
           and StuckB.  As you'd expect, StuckA will let
           gas through A regardless of the commanded set value,
           and likewise for StuckB. */
        if (StuckA) {
            OutA.p = In.p;
            OutA.t = In.t;
            In.mass_rate = OutA.mass_rate;
            OutB.p = 0.0;
            OutB.t = In.t;
        }

        if (StuckB) {
            OutB.p = In.p;
```

```
            In.mass_rate = OutB.mass_rate;
            OutA.p = 0.0;
            OutA.t = In.t;
        }
    }
}


/* Define the access functions to turn the valves on
   and off.  Note that there will be a probability of unsticking
   the valve when we try to open and close it.  When the valve
   is stuck open, we can't close it, and sending the command
   to open again might free the valve.  Also, when the valve
   is stuck closed, we can't open it, and so sending the
   command to close again might free the valve.  */
public void SetA() {
    interval clear;
    storeval(state, 0);
    if (StuckA) {
        clear = check_to_clear_fault(probClearFault);
        if (clear = 1) RemoveFault;
    }
}

public void SetB() {
    interval clear;
    storeval(state, 1);
    if(StuckB) {
        clear = check_to_clear_fault(probClearFault);
        if (clear = 1) RemoveFault;
    }
}

/* Define the access functions that allow the user to
   inject faults to the valve.  These are mutually exclusive,
   so SetStuckOpen will do nothing if we called a SetStuckClosed
   without clearing it (and vice versa). */
public void SetStuckA(interval probclear)  {
    unless (StuckB) {
      unless (StuckA) {
        do {
            hence StuckA;
        } until (RemoveFault);
        storeval(probClearFault, probclear);
      }
    }
}

public void SetStuckB(interval probclear) {
    unless (StuckB) {
      unless (StuckA) {
        do {
            hence StuckB;
        } until (RemoveFault);
        storeval(probClearFault, probclear);
      }
    }
}

public void ClearFault() {
    RemoveFault;
```

```
    public void CreateValveFault() {
        interval faulttype, faultprob;
        faulttype = check_for_fault(default_state, state);
        if( faulttype = 1.0 ) {
            faultprob = get_prob_to_clear();
            SetStuckA(faultprob);
        }

        if( faulttype = 2.0 ) {
            faultprob = get_prob_to_clear();
            SetStuckB(faultprob);
        }
    }
}

/*********************************************************
  CreateValveFault()
 *********************************************************/
/* Used to enable the faults for the solenoid valves.  Will call
   itself iteratively every n simulation seconds, and each time
   it calls itself there is a constant (low) probability that
   the valve will become stuck.  This results in the valve faults
   having a geometric probability distribution.  May not be
   realistic, but the real point is to be able to add faults
   to the valves in order to test the control software!
 */

//CreateValveFault(),  // Uncomment these lines to enable the random faults

void CreateValveFault() {
    SolenoidOnOffGasValve X;
    SolenoidSwitchingGasValve Y;

    /* Check for faults in on-off solenoid valves */
    forall X in class SolenoidOnOffGasValve {
        X.CreateValveFault();
    }

    /* Check for faults in switch solenoid valves. Same code as
        above, but done for a different valve name. */
    forall Y in class SolenoidSwitchingGasValve {
        Y.CreateValveFault();
    }

    when(time = eval(time)+1) CreateValveFault();


/*********************************************************
 Check Valve (Gas) object
 *********************************************************/

class CheckGasValve {
    private interval dia, kl, density, volume_flow, mass_flow, p;
    public CheckGasValve(Flow In, Flow Out,
            interval diameter, interval loss_coeff) {
        always dia=eval(diameter);
        always kl=eval(loss_coeff);

        /* A check valve is designed to only allow fluid to
           travel in one direction.  If the output pressure is
```

```
        equal to ). The check valves don't have faults assumed
        that they won't fail. */

      /* When the output pressure is less than the input pressure,
         the mass flowrate through the valve depends on the
         geometry (Kl) and the pressure difference on both
         sides.  */

      /* The valve plays the following role in the sim.  Essentially,
         it looks at the pressure difference and determines the flowrate
         through that branch of the system.  The pressure on
         either side of the valve must be defined by other simulation
         objects, usually tanks.  If the input pressure is greater than
         the output pressure, the valve then uses the pressure
         difference and the characteristics of the input tank to determine
         the mass flowrate through the valve.  If the output pressure
         is greater than the input pressure, the flowrate is set to 0.
         This mass flowrate number is then sent to both the upstream and
         the downstream objects, for use in mass (molar) balance equations.

         If the input pressure is 0, that is the signal that there
         is no flow to go through the system (an upstream valve is off).
         The outgoing molar flowrate is then set to 0.  This signal is
         really only used with the valves connected to flow controllers. */

      always {

          density := (In.p * In.mw) / (r*In.t) * 1000;  /* Set to kg/m^3*/
          p := In.p - Out.p;

         unless ( (In.p <= Out.p) || (In.p = 0.0) ) {
             volume_flow = (1/4) * PI * dia^2
                 *sqrtPt(2*(In.p - Out.p)*1.0e5/(density*kl) );
             mass_flow = (In.p*volume_flow*1000) / (r*In.t);
         }

         if ( (In.p <= Out.p) || (In.p = 0.0) ) {
             mass_flow = 0.0;
         }

         Out.t = In.t;
         In.mass_rate = mass_flow;
         Out.mass_rate = mass_flow;
      }
   }
}


#endif
```